The Abdus Salam International Centre for Theoretical Physics

United Nations Educational, Scientific and Cultural Organization

IAEA International Atomic Energy Agency

1967-6

Advanced School in High Performance and GRID Computing

3 - 14 November 2008

From Source Code to Executable: Preprocessing / Compiling / Linking Makefiles (Part I)

KOHLMEYER Axel

University of Pennsylvania
Department of Chemistry
231 South 34th Street
PA 19104 Philadelphia
U.S.A.

From Source Code to Executable: Preprocessing, Compiling, Linking, and Makefiles

ICTP Advanced School in High Performance and GRID Computing

Axel Kohlmeyer
Center for Molecular Modeling
ICTP, Trieste – Italy, 04 November 2008

the abdus salam international centre for theoretical physics

Penn
University of Pennsylvania

Overview / Compiler

- The pre-process/compile/link process: the individual steps in detail
- Preprocessing in C and Fortran
- The C-preprocessor, typical directives
- Compilers: Fortran 77/95, C, C++
  Vendors: GNU, Intel
- Special Compiler flags
- Special Linker flags, Utilities

Pre-process / Compile / Link

- Consider the minimal C program 'hello.c':

```
#include <stdio.h>
int main(int argc, char **argv)
{
        printf(  hello world\n  );
        return 0;
}
```

- What happens, if we do?:

```
> cc -o hello hello.c
```

(try: `cc -v -o hello hello.c`)

Step 1: Pre-processing

- Preprocessing will handle all '#' directives
  - File inclusion
  - Conditional compilation
  - Macro expansion
- In this case: `/usr/include/stdio.h` is inserted and pre-processed
- Only pre-processing with -E flag:
  `> cc -E -o hello.pp.c hello.c`
- Last part of hello.pp.c and hello.c identical

Step 2: Compilation

- Compiler converts high-level language
- Output machine specific text (assembly)
- Parses text (lexical + syntactical analysis)
- Test for correctness, language dialect
- Translate to internal representation (IR)
- Optimization (reorder, merge, eliminate)
- Replace IRs with assembly blocks
- Crucial step: Performance, Correctness
- (Cross-compiler for different platform)
- Try: > cc -S hello.c (and look at `hello.s`)

Compilation cont'd

```
.LC0:
        .string "hello world\n"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        leave
        ret
```

Optimized Compilation

```
.LC0:
        .string "hello world"
        .text
        .p2align 2,,3
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        xorl    %eax, %eax
        leave
        ret
```

Assembler / Linker

- Assembler (as) translates assembly to binary
  - Creates so-call object files

  Try: > `cc -c hello.c`

  Try: > `nm hello.o`

  ```
  00000000 T main
           U printf
  ```
- Linker (ld) puts binary together with startup code and required libraries
- Final step, result is executable.

  Try: > ld -o hello hello.o

Adding Libraries

- Example 2: exp.c

```
#include <math.h>
#include <stdio.h>

int main(int argc, char **argv)
{
     printf("exp(2.0)=%f\n", exp(2.0));
     return 0;
}
```

- > `cc -o exp exp.c`
- Does not work. `exp` is part of math library
- => `cc -o exp exp.c -lm`

Pre-processing in C and Fortran

- Pre-processing mandatory in C/C++
- Pre-processing optional in Fortran
- Fortran pre-processing often implicit via file name: name.F, name.F90, name.FOR
- Use -DSOME_SYS to conditionally compile
- Use -DDEF_ARR=200 to set defaults
- Use capital letters to signal a define
- Use -I/some/path to search for include files
- Legacy Fortran packages frequently do:
  /lib/cpp -C -P -traditional -o name.f name.F

C-Pre-processor directives

- #define MYVAL 100
- #undef MYVAL
- #if defined(MYVAL) && defined(__linux)
- #elif (MYVAL < 200)
- #else
- #endif
- #include “myfile.c”
- #include <myotherfile.c>
- #define SQRD(a) (a*a)

Compilers: GNU, PGI, Intel

- We will cover only C/C++, Fortran 77/95
- GNU: gcc, g++, g77(old), gfortran(new), g95
  - Free, C/C++ quite good, gfortran/g95 not bad
  - 'native' Linux compilers
  - Support for many platforms
  - Support for cross-compilation
- PGI: pgcc, pgCC, pgf77, pgf90
  - Commercial with trial, x86 and x86_64
- Intel: icc, icpc, ifort
  - Commercial with trial and non-commercial,
  - x86, ia64 (Itanium), EM64t (=x86_64/Opteron)

Common Compiler Flags

- Optimization: -O, -O0, -O1, -O2, -O3, -O4, ...
  - Compiler will try to rearrange generated code so it executes faster
  - High optimization will not always execute faster
  - High optimization may alter semantics
- Compile only: -c
- Preprocessor flags: -I/some/dir -DSOM_SYS
- Linker flags: -L/some/other/dir -lm
  -> search for libm.so/libm.a also in /some/dir
- Read the documentation for details

Special Compiler Flags: GNU

- -mtune=i686 -march=i386 (-mcpu sets both)
  optimize for i686 cpu, use i386 instruction set
- -funroll-loops
  heuristic loop unrolling (for floating point codes)
- -ffast-math
  replace some constructs with faster alternatives
- -fomit-frame-pointer
  use stack pointer as general purpose register
- -mieee-fp
  turn on IEEE754 compliance / comparisons

Special Compiler Flags: Intel

- -tpp6, set cpu type, v10 supports GNU style
- -pc64, set floating point rounding to 64-bit
- -ip, ipo, interprocedural optimization
- -axPW, generate SSE, SSE3 instructions
- -unroll, heuristic loop unrolling
- -fpp -openmp, turn on OpenMP
- -i-static, link compiler runtime statically
- -mp, force IEEE floating point handling
- -mp1, almost force IEEE floating point
- -fast, shortcut for -xP -O3 -ipo -no-prec-div

Special Compiler Flags: PGI

- -tp=px, -tp=amd64, -tp=x64, -tp=piv
  generate architecture specific code
- -pc=64
  set floating-point rounding mode to 64-bit
- -Munroll, loop unrolling,
- -Mvect, vectorization (loop scheduling)
- -fast, -fastsse, short cuts to optimization flags
- -mp for OpenMP support
- -Mipa, turn on interprocedural analysis
- -Kieee, turn on IEEE floating point

Linker Issues, Utilities

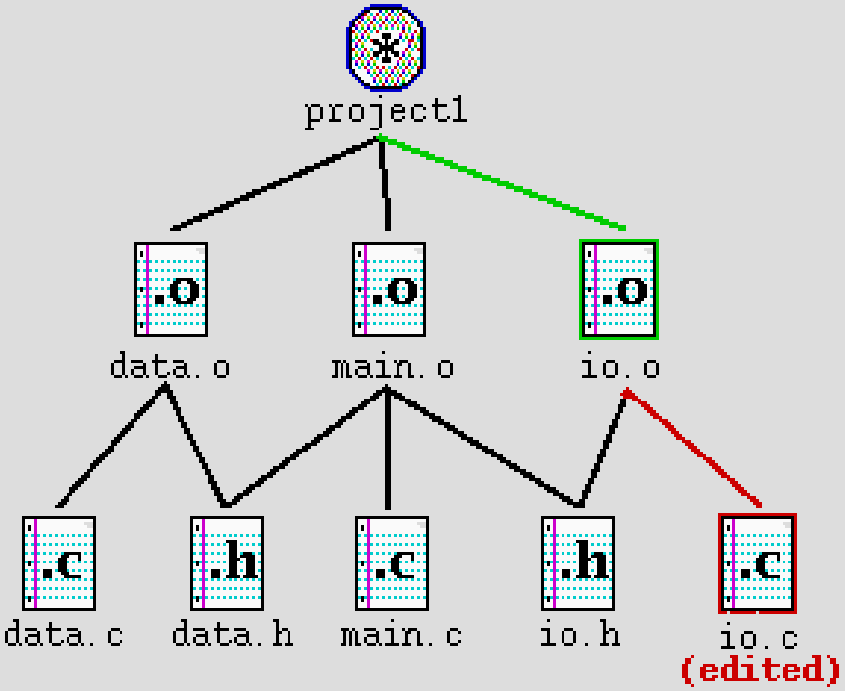
- Linux defaults to dynamic libraries:
  > ldd hello
  libc.so.6 => /lib/i686/libc.so.6
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2
- /etc/ld.so.conf, LD_LIBRARY_PATH
  define where to search for shared libraries
- Ifort -Wl,-rpath,/some/dir
  will encode /some/dir into binary for
  searching for dynamical libraries.
  Same for icc, GNU gcc/g++/gfortran

Overview / Make

- The idea behind make
- General Syntax
- Rules Examples
- Variables Examples
- Pattern Rules
- Special Rules
- Dependencies
- Conventions

Makefiles: Concepts

- Simplify building large code projects
- Speed up re-compile on small changes
- Consistent build command: make
- Platform specific configuration via Variable definitions



Makefiles: Syntax

- Rules:

```
target: prerequisites
            command
```

^this must be a 'Tab' (|<- ->|)

- Variables:

```
NAME= VALUE1 VALUE2 value3
```

- Comments:

```
# this is a comment
```

- Special keywords:

```
include linux.mk
```

Makefiles: Rules Examples

```
# first target is default:
all: hello sqrt

hello: hello.c
        cc -o hello hello.c

sqrt: sqrt.o
        f77 -o sqrt sqrt.o
sqrt.o: sqrt.f
        f77 -o sqrt.o -c sqrt.f
```

Makefiles: Variables Examples

```
# uncomment as needed
CC= gcc
#CC= icc -i-static
LD=$(CC)
CFLAGS= -O2

hello: hello.o
        $(LD) -o hello hello.o

hello.o: hello.c
        $(CC)-c $(CFLAGS) hello.c
```

Makefiles: Automatic Variables

```
CC= gcc
CFLAGS= -O2

howdy: hello.o yall.o
        $(CC) -o $@ $^

hello.o: hello.c
        $(CC)-c $(CFLAGS) $<

yall.o: yall.c
        $(CC)-c $(CFLAGS) $<
```

Makefiles: Pattern Rules

```
OBJECTS=hello.o yall.o

howdy: $(OBJECTS)
        $(CC) -o $@ $^

hello.o: hello.c
yall.o: yall.c

.c.o:
        $(CC)-o $@ -c $(CFLAGS) $<
```

Makefiles: Special Targets

```
.SUFFIXES:
.SUFFIXES: .o .F

.PHONY: clean install

.F.o:
     $(CPP) $(CPPFLAGS) $< -o $*.f
     $(FC)-o $@ -c $(FFLAGS) $*.f

clean:
       rm -f *.f *.o
```

Makefiles: Calling make

- Override Variables:
- `make CC=icc CFLAGS='-O2 -unroll'`
- Dry run (don't execute):
- `make -n`
- Don't stop at errors (dangerous):
- `make -i`
- Parallel make (requires careful design)
- `make -j2`
- Alternative Makefile
- `make -f make.pgi`

Makefiles: Building Libraries

```
ARFLAGS= rcsv
LIBOBJ= tom.o dick.o harry.o

helloguys: hello.o libguys.a
     $(CC) -o $@ $< -L. -lguys

libguys.a: $(LIBOBJ)
     ar $(ARFLAGS) $@ $?

tom.o: tom.c guys.h
dick.o: dick.c guys.h
harry.o: harry.c guys.h
hello.o: hello.c guys.h
```

Makefiles: Automatic Dependencies

```
LIBSRC= tom.c dick.c harry.c
LIBOBJ= $(LIBSRC:.c=.o)
LIBDEP= $(LIBSRC:.c=.d)
.c.d:
    $(CC) $(CFLAGS) -MM $< > $@

include $(LIBDEP)
```

alternatively (note, some makes require .depend to exist):

```
.depend dep:
    $(CC) $(CFLAGS)-MM $(LIBSRC) > .depend

include .depend
```

Makefile Portability Caveats

- Always set the SHELL variable:
  `SHELL=/bin/sh`
  (make creates shell scripts from rules).
- GNU make has many features, that other make programs don't have and vice versa
- Use only a minimal set of Unix commands: cp, ls, ln, rm, mv, mkdir, touch, echo,...
- Implement some standard 'phony' targets: all, clean, install